Parallel Processing Letters
© World Scientific Publishing Company

# PROOF OF A REAL-TIME MUTUAL-EXCLUSION ALGORITHM

J. ALLEN CARRUTH
and
JAYADEV MISRA
*Department of Computer Sciences*
*University of Texas at Austin*
*Austin, Texas 78712, USA*

Received ??/??/??
Revised ??/??/??
Communicated by

ABSTRACT

Michael Fischer has proposed a mutual exclusion algorithm that ingeniously exploits real time. We prove this algorithm using the time-honored technique of establishing an appropriate invariant.

*Keywords:* Real-time, mutual exclusion, formal verification

## 1. Introduction

Michael Fischer[1] has proposed a mutual exclusion algorithm that ingeniously exploits real time. We prove the correctness of this algorithm using only the fact that time never runs backwards. Other important facts about time — that eventually time increases beyond any bound — are unnecessary for this proof.

The structure of the proof follows the usual pattern of suggesting an invariant, verifying that the suggested invariant is indeed an invariant and showing that the invariant implies mutual exclusion. The invariant is, as usual, a state predicate. We introduce some auxiliary variables that simplify reasoning about time: For a state-predicate $p$, let $\overline{p}$ denote the last value of time at which $p$ became *true*. We call $\overline{p}$ the *punch* of $p$ (the time at which $p$ last *punched* the clock). We specify the timing constraints of the algorithm succinctly using such variables.

## 2. Informal Description of the Algorithm

There are $N$ processes, numbered 1 through $N$, and a global variable $x$ that assumes an integer value between 0 and $N$. Figure 1 shows the state transitions of process $i$, $1 \leq i \leq N$. A process transits from $e$ to $a$ to wait for entry to its critical section. The edges of the other transitions are labeled with either an assignment,

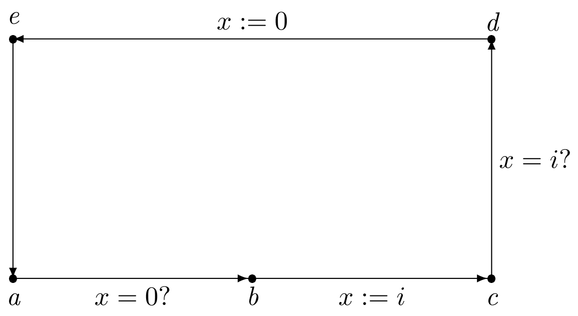

Fig. 1. The state transitions of a process $i$, $1 \leq i \leq N$.

$x := i$ or $x := 0$, or a test, $x = 0?$ or $x = i?$. An assignment on an edge denotes that the state transition is accompanied by an assignment of the corresponding value to $x$. A test on an edge denotes that the transition takes place only if the test succeeds. Process state is $d$ when it is in the critical section. Assume that all tests and assignments are atomic. Initially, all processes are in states $e$ and $x = 0$.

It is easy to construct a scenario where two processes are in their critical sections simultaneously. Timing constraints, given below, guarantee that this possibility is avoided.

(T1) Transition from $b$ to $c$ is completed within a unit of time. Observe that this transition only requires assigning a value to $x$, and, therefore, the transition is entirely within the control of a process.

(T2) Transition from $c$ to $d$ takes more than one unit of time. This requirement is implemented by process $i$ waiting for more than a unit of time before testing $x = i?$. Observe that this transition may never complete.

We will show that mutual exclusion is now guaranteed, i.e., two different processes are never in their $d$-states simultaneously.

**Remark:** There is no requirement, as yet, that a process transit out of its $d$ or $e$-state. Thus, a process may stay forever in $e$ (i.e., never attempting to enter its critical section) or in $d$, thereby preventing all other processes from entering their critical sections forever. □

## 3. Formal Description of the Algorithm

Let $s_i$ denote the state of process $i$; $s_i$ takes values from $\{a, b, c, d, e\}$. The initial state of the system is

**initially** $(\forall\ i\ \ ::\ \ s_i = e)\ \wedge\ x = 0$

The state transitions of process $i$ are given by

```
   {$\alpha_i$} $s_i$ := $a$        if $s_i = e$
[] {$\beta_i$} $s_i$ := $b$        if $s_i = a \ \wedge\ x = 0$
[] {$\gamma_i$} $s_i, x$ := $c, i$   if $s_i = b$
[] {$\delta_i$} $s_i$ := $d$        if $s_i = c \ \wedge\ x = i$
[] {$\epsilon_i$} $s_i, x$ := $e, 0$   if $s_i = d$
```

There is no fairness requirement on the executions of these statements. Executing a statement in a state where its guard does not hold — such as executing $\beta_i$ when $x \neq 0$ — causes no state change; any execution of a statement when its guard is *true* is called an *effective execution.*

**Notation:** We will use the following abbreviations

$$a_i \ \equiv\ s_i = a \qquad b_i \ \equiv\ s_i = b \qquad , \ \ldots\ , \qquad e_i \ \equiv\ s_i = e$$

Observe that these predicates are mutually exclusive, i.e.,

$$a_i \ \wedge\ b_i \ \equiv\ \mathit{false},\ \text{etc.},$$

and $a_i \ \vee\ b_i \ \vee\ c_i \ \vee\ d_i \ \vee\ e_i \ \equiv\ \mathit{true}$ □

### *3.1. Formalization of Time*

In order to state the timing constraints, we introduce a variable *now* [2]. Informally, the value of *now* at any point during the computation is the current time. The value of *now* is changed by some mechanism outside the given program; the mutual exclusion program can read the current time and assign it to a variable $t$ by executing

$$t \ := \ \mathit{now}$$

The mechanism (or process) that changes *now* could operate synchronously or asynchronously with the actions of the given program. Thus, the value of *now* before and after the execution of

$$t \ := \ \mathit{now}$$

may be different (denoting that execution of this statement consumes some time). The value assigned to $t$ in this case is the value of *now* just before the execution of this statement is started. This interpretation supports the axiom of assignment: Predicate $p(t)$ holds after this assignment if $p(\mathit{now})$ holds before.

For this paper, we require only that (1) *now* assumes non-negative real values and (2) *now* is monotone nondecreasing. For a formal basis for the introduction of time, including the requirement about the eventual increase of *now*, see Abadi and Lamport [2].

It is convenient to introduce the following auxiliary variables for study of real time systems. For a state predicate $p$, let $\overline{p}$ be the value of *now* when $p$ last became *true* (more precisely, $\overline{p}$ is the value of *now* just prior to the execution of the action that last truthified $p$); initially $\overline{p}$ equals *now* if $p$ holds, else $\overline{p} < 0$. This definition of $\overline{p}$ can be expressed directly as a property of the program, or $\overline{p}$ can be defined by augmenting a program text, as shown below; both definitions are equivalent. We introduce three such variables, $\overline{b}_i, \overline{c}_i, \overline{d}_i$ , by augmenting $\beta_i, \gamma_i, \delta_i$,

$$\begin{array}{lll}
\{\beta_i\}\ s_i, \overline{b}_i \ := \ b, now & \text{if} & s_i = a \ \wedge \ x = 0 \\
\{\gamma_i\}\ s_i, x, \overline{c}_i \ := \ c, i, now & \text{if} & s_i = b \\
\{\delta_i\}\ s_i, \overline{d}_i \ := \ d, now & \text{if} & s_i = c \ \wedge \ x = i
\end{array}$$

Initially, $\overline{e}_i = now$ and $\overline{b}_i, \overline{c}_i, \overline{d}_i$ are negative. From the fact that *now* is non-negative and monotone nondecreasing, we can derive, for any $p$,

(Observation 1) $\overline{p} \leq now$.

**Remark:** The auxiliary variables $\overline{p}$ can be used to state the most common kinds of real-time constraints:

Once $p$ becomes *true* it remains *true* for at least $\Delta$ units,

can be written as

$$\neg p \ \wedge \ \overline{p} \geq 0 \ \Rightarrow \ now > \overline{p} + \Delta$$

and, $p$ is falsified within $\tau$ units of being *true*, is expressed by

$$p \ \Rightarrow \ now \leq \overline{p} + \tau \qquad \square$$

*3.2. Timing Constraints*

We can now state (T1,T2) formally. For all $i$, $1 \leq i \leq N$,

(T1) $(c_i \ \vee \ d_i) \ \Rightarrow \ \overline{c}_i \leq 1 + \overline{b}_i$

(T2) $d_i \ \Rightarrow \ 1 + \overline{c}_i < \overline{d}_i$

The antecedent of (T1), $c_i \ \vee \ d_i$, guarantees that in the current state both $\overline{b}_i$ and $\overline{c}_i$ are defined; similar remarks apply for the antecedent of (T2).

## 4. Proof of Mutual Exclusion

We establish the following two predicates as invariants. In the following, $j, k$ satisfy $1 \leq j \leq N$ and $1 \leq k \leq N$.

(I1) $(\forall\ j, k \ :: \ x = k \ \Rightarrow \ \overline{b}_j \leq \overline{c}_k)$

(I2) $(\forall\ k \ :: \ d_k \ \Rightarrow \ x = k)$

Mutual exclusion is immediate from I2:

$$d_i \;\wedge\; d_j \;\Rightarrow\; x = i \;\wedge\; x = j \;\Rightarrow\; i = j$$

Next, we prove that for the program of Section 3 augmented with the timing constraints, the predicates (I1,I2) are invariants.

**Note:** To be completely formal, we should also show that (I1,I2) cannot be falsified by the process that changes *now*. Since *now* does not appear in either predicate, this demonstration is trivial. $\Box$

*4.1. Proof of the invariance of (I1)*

We rewrite (I1) as $(\forall\, j,k \;::\; x \neq k \vee \overline{b}_j \leq \overline{c}_k)$ to simplify logical manipulations. Initially, $x = 0$. Therefore, initially $x \neq k$, for any $k$, $1 \leq k \leq N$, and, hence, (I1) holds initially. Next, consider the actions that can falsify the terms in (I1), for arbitrary $j, k$.

- $x \neq k$ can only be falsified by setting $x$ to $k$, i.e., by effectively executing $\gamma_k$, which is

  $$\{\gamma_k\}\; s_k, x, \overline{c}_k \;:=\; c, k, \mathit{now} \qquad \text{if} \quad s_k = b$$

  To see that this action establishes $\overline{b}_j \leq \overline{c}_k$ as a postcondition we have to show, using the axiom of assignment to replace $\overline{c}_k$ by *now*, that $\overline{b}_j \leq \mathit{now}$ is a precondition. This follows from Observation 1. Therefore, $\gamma_k$ preserves (I1).

- the term $\overline{b}_j \leq \overline{c}_k$ can be affected only by the actions $\beta_j$ (that may change $\overline{b}_j$) and $\gamma_k$ (that may change $\overline{c}_k$). We have shown above that $\gamma_k$ preserves (I1). We now show that $\beta_j$ also preserves (I1). A precondition for the effective execution of $\beta_j$ is $x = 0$, and $\beta_j$ preserves $x = 0$. Therefore, $x = 0$, i.e., $x \neq k$ is a postcondition of an effective execution of $\beta_j$.

*4.2. Proof of the invariance of (I2)*

Initially $\langle \forall\, k \;::\; e_k \rangle$. Therefore, (I2) holds initially. Next, consider the actions that can falsify $\neg d_k \;\vee\; x = k$, for arbitrary $k$.

- $\neg d_k$ can be falsified only by setting $s_k$ to $d$, i.e., by effectively executing $\delta_k$. A precondition for the effective execution of $\delta_k$ is $x = k$. The action $\delta_k$ does not assign to $x$, and, hence, preserves $x = k$. Therefore, $\neg d_k \;\vee\; x = k$ holds as a postcondition of $\delta_k$.

The predicate $x = k$ can be falsified by (1) setting $x$ to 0, i.e., executing $\epsilon_i$, for some $i$, or (2) setting $x$ to $i$, $i \neq k$, i.e., executing $\gamma_i$, $i \neq k$. We consider these two possibilities, next.

- Executing $\epsilon_i$, for some $i$: Action $\epsilon_i$ has a precondition $d_i$. Then using (I2), for $k \neq i$, $\neg d_k$ holds as a precondition; also, $\neg d_k$ is preserved by $\epsilon_i$. Furthermore, $\neg d_i$ is a postcondition of $\epsilon_i$. Therefore, $\neg d_k$ holds as a postcondition of $\epsilon_i$, for any $i$.

- Executing $\gamma_i$, $i \neq k$: We show that the effective execution of $\gamma_i$ preserves (I2), i.e., if (I2) holds prior to the execution of $\gamma_i$ (I2) also holds after the completion of $\gamma_i$.

  For this proof, we have (I1) as a precondition, and we have assumed (I2) to be a precondition. The text of $\gamma_i$ is,

  $$\{\gamma_i\}\ s_i, x, \overline{c}_i \ := \ c, i, \mathit{now} \qquad \text{if} \quad s_i = b$$

  From this text, a postcondition of $\gamma_i$ is $c_i$, and using (T1) we have $\overline{c}_i \leq 1 + \overline{b}_i$ also as a postcondition. Summarizing, we are given for $\gamma_i$

  precondition: (I1) and (I2)
  postcondition: $\overline{c}_i \leq 1 + \overline{b}_i$

  and we have to show for $\gamma_i$

  postcondition: (I2)

  In order to demonstrate that (I2) is a postcondition of $\gamma_i$, it is sufficient to show, using the axiom of assignment, that $d_k \Rightarrow i = k$ is a precondition for arbitrary $k$; we show the stronger result that $\neg d_k$ is a precondition of $\gamma_i$, or equivalently, $d_k \Rightarrow \neg d_k$.

  $$\begin{array}{ll} & d_k \\ \Rightarrow & \quad \{ \text{ from I2 } \} \\ & x = k \\ \Rightarrow & \quad \{ \text{ from I1, using } i \text{ for } j \ \} \\ & \overline{b}_i \leq \overline{c}_k \\ \Rightarrow & \quad \{ \text{ arithmetic } \} \\ & 1 + \overline{b}_i \leq 1 + \overline{c}_k \\ \Rightarrow & \quad \{ \ \mathit{now} \leq 1 + \overline{b}_i, \text{ apply the axiom of assignment to the postcondition of } \gamma_i\text{: } \overline{c}_i \leq 1 + \overline{b}_i \ \} \\ & \mathit{now} \leq 1 + \overline{c}_k \\ \Rightarrow & \quad \{ \ \overline{d}_k \leq \mathit{now}, \text{ from observation 1 } \} \\ & \overline{d}_k \leq 1 + \overline{c}_k \\ \Rightarrow & \quad \{ \text{ from T2 } \} \\ & \neg d_k \end{array}$$

*4.3. The role of the timing constraints*

The structures of the proofs make it clear where and why the timing constraints, (T1) and (T2), are required. Both of them are needed to show that execution of $\gamma_i$ preserves (I2), that the $i^{th}$ process is prevented from setting $x$ (to $i$) if some process is in its critical section. The crucial part in the proof that uses (T1) is the assertion that $now \le 1 + \bar{c}_k$ is a precondition for the execution of $\gamma_i$, i.e., $\gamma_i$ executes within one unit of the effective execution of $\gamma_k$, for any $k$. The proof step that exploits (T2) is $(\bar{d}_k \le 1 + \bar{c}_k) \Rightarrow \neg d_k$, i.e., any process $k$ takes at least a unit of time to transit to state $d$, its critical section.

**Acknowledgement:** We are grateful to Jacob Kornerup for suggestions about the structure of the proof. Leslie Lamport has raised a number of issues about real-time programming that go considerably beyond what has been attempted in this paper.

1. M. J. Fischer. Personal communication with Leslie Lamport, 1985.
2. Martin Abadi and L. Lamport. An old fashioned recipe for real time. *TOPLAS*, 16(5):1543–1571, Sep. 1994.